Language as Social Software[1]

Rohit Parikh

Introduction: Computer science has generated much interest among philosophers in recent years, in part because of the existence of computer models of the mind, but also because of the relevance of Church's Thesis[2] to the discussion about the possibilities of artificial intelligence. Much of the sort of computer science invoked in these discussions, however, is fairly old, going back to the work of Church, Turing and Gödel. Apart from connectionism, most recent work in computer science, especially the work on program correctness, compilers, and the semantics of programming languages has not influenced philosophical discussion. Perhaps philosophers see these topics as too technical, of interest only to computer scientists and having little or no bearing on philosophical issues. What I intend to show here is that, to the contrary, this work too, beginning with Floyd and Hoare, and developed by others like Pratt, Pnueli, Scott, Milner, and myself, can yield significant insights into philosophical issues.

In part I of this paper I briefly describe some recent developments in computer science, especially in the areas of program correctness, dynamic logic and temporal logic. In part II, I use analogies from computer science to approach contemporary philosophical puzzles such as Quine's indeterminacy thesis.[3] Searle's Chinese room puzzle,[4] and the Wittgenstein-Kripke plus-quus puzzle.[5] Part I is somewhat technical, but it can be followed with no more than high school algebra and elementary logic. Part II is intended to be fairly self-contained, so the reader who so desires may skim part I or even skip directly to part II.

The principal tool of part II is an analogy between: (1) the denotational semantics of a high level programming language versus the operational semantics of the machine language; and (2) a meaning theory for a natural language versus the actual linguistic and non-linguistic behavior of the members of a speech community. I try to use this analogy to provide computer science counterparts for the various philosophical puzzles mentioned above. My hope is that these computer science counterparts will prove somewhat easier to understand, or at least to become clear about, and that this understanding can help to clarify the philosophical puzzles. The view of language developed in part II is not primarily truth-theoretic but, as I see it, more Wittgensteinian. Language is thought of as part of the life of a community and its purpose is seen as enabling the community to function more effectively. Inasmuch as the conveying of information needs to be accurate (in some sense) to be useful, there is here an underlying notion of truth. But there are large parts of language which are not informational, for example, commands, exclamations, as well as performatives. Moreover, though there is and must be a great deal of overlap in the way in which different members of a community use language, still different individuals in the same community, or even the same individual on different occasions, cannot, as I have argued elsewhere,[6] use language in literally the same way. An example of this are the uses of vague predicates. While attempts to specify truth-conditions and a logic for vague predicates have been notoriously unsuccessful, pragmatic approaches that do not assume uniformity in the use of language seem to work quite well.

[1]I am indebted to Arthur Collins, Juliet Floyd, Konstantinos Georgatos and Larry Moss for comments on previous versions of this paper. Research supported by a grant from the CUNY-FRAP program.

[2]The philosophical thesis that every effectively computable function is recursive.

[3]See W. V. Quine, *Word and Object* (Cambridge, Mass.: M.I.T. Press, 1960), chapter 2.

[4]See John Searle, "Minds, Brains and Programs," *Behavioral and Brain Sciences* 3 (1980): 417-457.

[5]See Saul Kripke, *Wittgenstein on Rules and Private Language* (Cambridge, Mass.: Harvard University Press, 1982).

[6]"Vagueness and Utility: The Semantics of Common Nouns," *Linguistics and Philosophy* 17 (1994): 521-35.

The reader will thus not go far wrong by thinking of the development in part II as a formal version of what Wittgenstein called "language games." [7]

I

Program Correctness.

Computer programmers write programs for specific purposes and not just to pass the time. The question thus arises at once, for any given program, whether it has fulfilled its purpose. For example, a spell-checking program should take a document, say this very paper, and produce a list of misspelt words. The program then fulfills its purpose, or is *correct,* if it produces all and only words which are misspelt. If it produces words that are *not* misspelt, or fails to produce some that *are*, then it is not correct.

But how do we *know* that a given program is correct? The way which most of us (including software companies like Microsoft) proceed in practice is to try the program and see if it works. If a program has worked satisfactorily for us a few times, we tend to feel secure and assume that it is correct. The difficulty with this procedure–called *testing–* is that a program may work correctly some of the time or even much of the time but not *all* of the time, in which case it may be risky to use it on a new input. As Dijkstra has remarked in this context, testing can only prove that an incorrect program is incorrect but not that a correct program is correct.[8] Those of us who regard programs as part of the natural world, and hence feel that falsifiability is all we can hope for, may think that this is all that one should want.

But not all have been satisfied by mere falsifiability. Since programs are human artifacts, one might think that there must be a better way, namely, to *prove* them to be correct. Mathematical techniques for this purpose were developed by Floyd and converted to formal logics by Hoare, Pratt, and others.[9] Let us see by means of a simple example how this works.

Let α_1 be the program $(x := 2); (y := x \times z)$. Here an instruction of the form "$x := t$" means, set the variable x to the (current) value of the term t (here 2) and the symbol ";" stands for "and then." So α_1 is the program that first sets x to be the integer 2 and then sets y to be equal to x times z. How do we talk about the properties of α_1?

Hoare's notation for expressing properties of programs, which he uses in his logic, is $\{A\}\alpha\{B\}$ where A and B are ordinary first order formulas and α is some program. The meaning of the statement $\{A\}\alpha\{B\}$ is that, if the formula A is true when the program α starts, then the formula B will be true when it finishes. We can now show that the formula $\{z{=}4\}\alpha_1\{y{=}8\}$ *must* hold, and we need not resort to any testing. For clearly, given that z is 4, after the step $x{:=}2$, z will still be 4 and x must be 2 since it has just been set to that value. Now the next step sets y to $x{\times}z$. But since x is 2 and z is 4, $x{\times}z$ must be 8 and hence y is indeed set to 8. For any other values of z, it is clear that executing α_1 always results in $y{=}2{\times}z$ being true.

[7]In *Philosophical Investigations*, 3rd edn., eds. G. E. M. Anscombe and R. Rhees, trans. G. E. M. Anscombe (Oxford: Basil Blackwell and Mott, 1958).

[8]Edsger Dijkstra, *A Discipline of Programming* (New York: Prentice Hall, 1976), p. 20.

[9]Some good surveys of the field are: Dexter Kozen and Rohit Parikh,"An Elementary Completeness Proof for PDL," *Theoretical Computer Science* 14 (1981): 113-118, Dexter Kozen and Jerzy Tiuryn, "Logics of Programs," in *Handbook of Theoretical Computer Science*, vol. B, ed. Jan van Leeuwen (Cambridge: Mass.: M.I.T. Press/Elsevier, 1990), pp. 789-840, Rohit Parikh, "The Completeness of Propositional Dynamic Logic," *7th MFCS, Springer Lecture Notes in Computer Science* 64 (1978): 403-415, Amir Pnueli, "The Temporal Logic of Programs," *Proceedings of the 18th Annual IEEE Symposium on the Foundations of Computer Science* (1977): 46-57, and Allan Emerson, "Temporal and Modal Logic," in *Handbook of Theoretical Computer Science*, vol. B, ed. Jan van Leeuwen (Cambridge, Mass.: M.I.T. Press/Elsevier, 1990) pp. 995-1072.

The above argument made implicit use of one of Hoare's rules: if some program α is the same as $\beta;\gamma$ and if the properties $\{A\}\beta\{B\}$ and $\{B\}\gamma\{C\}$ hold, then $\{A\}\alpha\{C\}$ must hold. In this example, β is (x:=2), and γ is (y:=$x\times z$). Also A is z=4, B is z=4$\wedge x$=2, and C is y=8. It is evident that both $\{A\}\beta\{B\}$ and $\{B\}\gamma\{C\}$ hold.

A more sophisticated Hoare rule deals with the construct "*while–do.*" If β is a program we already have, then the program "*while A do β*" consists of repeatedly executing the program β as long as the condition A holds, and it terminates only when A is no longer true. The cooking instruction "stir until the sauce is thick" is of this form. It is (roughly) equivalent to, "*while* the sauce is not thick, *do* stir." Thus if β is the program "stir once," and A is "the sauce is not thick," then the instruction has the form "while A do β." In general, "*while A do β*" means, "check if A is true and if it is, then do β once. Repeat until A is no longer true." Of course if the sauce never gets thick, then you never stop stirring!

The Hoare rule (slightly simplified) for such a case is: If α is "*while A do β*" and $\{B\}\beta\{B\}$ holds, then derive $\{B\}\alpha\{B\wedge\neg A\}$. In other words, provided that β preserves the truth of B and if B holds when α begins, then B will still hold when α ends, and moreover, A will be false. Suppose B is the formula "the baby needs changing," then provided that B is left invariant by one stirring, and provided that the baby needed to be changed at the beginning of the stirring, then, by Hoare's rule, at the end of all the stirrings the sauce will be thick (that is, not non-thick), and the baby will (still) need changing.

To take a somewhat more interesting example, consider the program α_g for computing the greatest common divisor of two positive integers. It is given by

$$(x := u); (y := v); (\text{while } (x \neq y \text{ do } (\text{if } x < y \text{ then } y := y - x \text{ else } x := x - y))).$$

The program sets x, y to u, v respectively and then repeatedly subtracts the smaller of x, y from the larger until the two numbers become equal. If u, v are positive integers then this program terminates with $x = gcd(u,v)$, the greatest common divisor of u, v. The reason is that after the initial (x:=u);(y:=v) clearly $gcd(x,y) = gcd(u,v)$. Now both the instructions $x := x - y$ and $y := y - x$ leave the gcd of x, y unchanged. Thus if B is $gcd(x,y) = gcd(u,v)$, and β is ($if\ x < y\ then\ y := y - x\ else\ x := x - y$), then $\{B\}\beta\{B\}$ holds. Thus if the program α_g terminates, then by Hoare's rule for "*while*," $x \neq y$ will be false, i.e. $x = y$ will hold, and moreover B will hold. Since $gcd(x,x) = x$, we will now have $gcd(u,v) = gcd(x,y) = gcd(x,x) = x$.

There is a feature of Hoare's logic which makes it logically incomplete. Suppose that α is $\beta;\gamma$, $\{A\}\alpha\{C\}$ holds, and we want to derive this fact from properties of β and γ. Then to use Hoare's rule, we need to find a B such that $\{A\}\beta\{B\}$ and $\{B\}\gamma\{C\}$ are both true. It turns out that, although such a B must always exist, it may not be expressible in first-order logic, even though both A and C are first-order formulae. That is to say, there may be no first-order description of the state of affairs that prevails when β has ended and γ has yet to begin.

Dynamic Logic.

Pratt's solution to the inexpressibility problem is to extend the language of first-order logic by allowing program modalities $[\alpha]$. $[\alpha]B$ means that B will hold if and when α terminates. Thus $\{A\}\alpha\{B\}$ holds if and only if A implies $[\alpha]B$. Also, to prove $\{A\}\beta;\gamma\{C\}$, the formula $[\gamma]C$ will work as the intermediate B whose absence was the problem for Hoare's rules described above. The argument for this last claim goes as follows

First, suppose that $\{A\}\beta;\gamma\{C\}$ is true. Suppose also that A is true in some state s and we do β to reach a new state t. Clearly if we *now* did γ, C would hold. So t satisfies $[\gamma]C$. But s was an arbitrary state satisfying A. Hence $\{A\}\beta\{[\gamma]C\}$ holds. Second, $\{[\gamma]C\}\gamma\{C\}$ holds by the very meaning of $[\gamma]C$. For, if t satisfies $[\gamma]C$, then we know that if we did do γ we would reach a state in which we would have C. Hence if we do γ, we will evidently have C. Thus, if we take B to be $[\gamma]C$, then both $\{A\}\beta\{B\}$ and $\{B\}\gamma\{C\}$ hold. Since both $\{A\}\beta\{[\gamma]C\}$ and $\{[\gamma]C\}$ $\gamma\{C\}$hold, we can indeed derive $\{A\}\beta;\gamma\{C\}$.

Although first-order dynamic logic is highly undecidable, its propositional version, propositional dynamic logic (PDL) can be effectively axiomatized using the Segerberg axioms, a fact shown independently by Gabbay and by me. Dynamic logic also allows us to express dispositions. For example, an object is *fragile* if when thrown, it will break. So we can write $\mathit{Fragile}(x) : [\mathit{thrown}]$ $\mathit{Broken}(x)$. As Thorne McCarty has pointed out,[10] dynamic logic is appropriate for formalizing notions of permission and obligation, since the permissible and the obligatory characterize actions rather than propositions. Thus dynamic logic may well have a domain of applications even larger than foreseen.

An extension of dynamic logic is game logic, which can be used to show that many-person interactions can have certain desirable properties, for example, that there exists a fair algorithm for sharing something among n people.[11]

Suppose that n thieves have stolen a cake and, following the dictum of "honor among thieves," want to divide it fairly among one another. One algorithm for this scenario goes as follows: thief number one cuts out a piece which she claims is her fair share. Then the other thieves look at it in turn, and any thief who thinks it is too big may reduce it. After all the thieves have looked at it, if no one has reduced it, then thief number one takes the piece. If it has been reduced, then the last one who did so takes the piece. In either case, we now have $n - 1$ thieves to provide for, and we can repeat the algorithm.

Using game logic one can show that this algorithm is fair in the sense that every thief has a strategy whereby she can get $1/n$ of the cake regardless of what the other thieves do. The strategy is: when asked to pick, pick exactly $1/n$. If asked to look at a piece that someone else has picked, then let it pass if it is at most $1/n$. If it is more than $1/n$, reduce it to $1/n$.

This case serves as an example of a program involving many different agents or computers, unlike the program α_g, which involves only one. Such multi-agent programs or procedures occur constantly in social life, from traffic regulations to rules governing getting a building permit; and the study of their correctness and efficiency is therefore of the utmost importance.

Temporal Logic.

Temporal logic, an alternative to dynamic logic followed by Pnueli and others, abandons modularity (or compositionality), the principle (implicit in Hoare's rules) that we should derive the properties of a program from those of its sub-parts. For instance, the Hoare rule for ";" derived the properties of $\beta;\gamma$ from the properties of β and γ. Temporal logic, by contrast, reasons about one program at a time, focusing on the passage of time as a program runs. Thus in temporal logic we use operators such as "the property A will hold sometime in the future" or "A will hold until B does." The structure of time may be assumed to be linear or branching. The latter case arises if the course of the program is not determined but depends on random events like coin tosses or on external influence.

II

[10] L. Thorne McCarty, "Permissions and Obligations," in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, ed. Alan Bundy (Los Altos: William Kaufmann Inc., 1983), pp. 287-294.

[11] Rohit Parikh "The Logic of Games," *Annals of Discrete Mathematics* 24 (1985): 111-140.

Specification and implementation.

Clearly every program does *something* and therefore every program is correct if we are willing to accept whatever it does. Generally, however, the program was written for some purpose, and so we have some prior idea of what we want it to do. If this idea can be made precise then it is called the *specification* of the program. A program is correct only relative to a specification, that is, it is correct if what it does is what the specification demands that it do. Our little α_1 of part I was correct with respect to the specification *make y equal to 2 times z* and incorrect with respect to the specification *make y equal to 4 times z.*

Compilers and their correctness.

Computers come with their own mother tongue, what is usually called *machine language.* The commands in machine language tend to be rather primitive, like "store the contents of register x in register y," or "increment register y by 1." Programmers, however, rarely write their programs in machine language. Rather, they tend to write in what is called a high-level language. Such a language may contain complex commands like "sort the given list of numbers in increasing numerical order," or "change all occurrences of 'Dreben' to 'Burt' in the given document." Since the programmer writes his programs in one language and the machine understands another, some kind of translation mechanism is needed, and this mechanism–which is itself a program–is called the *compiler* for the particular high level language. The compiler takes a high level program α produced by the programmer and translates it into a machine language program $\beta{=}C(\alpha)$.

Whenever a high-level program α runs on a computer, it is first compiled. This process converts α into the machine language program $C(\alpha)$, the results of whose execution are then interpreted as outputs of α. There is an assumption that the operation of the compiled program mimics the mathematical properties of the source program α. We shall come back to this important assumption.

A high level programming language L has a "denotational semantics" corresponding to the usual model-theoretic semantics for a formalized language; at the denotational level, α refers to an abstract mathematical object, perhaps a function. The machine language, by contrast, has an "operational semantics," which is essentially a description of what happens inside the computer when a compiled program is run. There is an analogy between: (1) the denotational semantics of a high-level programming language versus the operational semantics of the machine language and (2) a meaning theory for a natural language versus the actual linguistic *and* non-linguistic behavior of the members of the community of users of this language. In *Figure* 1 below, *DS* stands for denotational semantics, *OS* for operational semantics, *MT* for a meaning theory for a natural language, and *Beh* for linguistic and non-linguistic behavior. The left side of the diagram is the computer science side, the right side is the natural language side.

$$\begin{tikzcd}
DS \arrow[r, leftrightarrow] \arrow[d, dash] & MT \arrow[d, dash] \\
OS \arrow[r, leftrightarrow] & Beh
\end{tikzcd}$$

Figure 1

In what follows, we will, as promised, try to use this analogy to sharpen some issues involving meaning, translation, and rule-following. What does it now mean to say that a compiled program is correct with respect to some given specification? Clearly, if the compiler C is badly written, then, even if the program α is correct, the compiled program $C(\alpha)$ might not be. So we must ask that for a compiler to be correct it should transform correct high-level programs into correct machine language programs. This becomes a requirement we will impose before accepting the compiler. But to enforce this requirement, we need a precise criterion of what it amounts to for a compiled program to be correct.

We note now that the compiler is generally written by a person who will not know the specification which the programmer will have in mind in a particular case. A similar situation arises when someone writes a translation manual from English to German. To get to the station, I must formulate the right question in English–say, "Where is the station?" The translation manual yields a corresponding question in German, for example, "Wo ist der Bahnhof, bitte?" If we use the question that we got from the manual, there will, hopefully, then be some linguistic response, which, when converted from German back to English, will provide me with the directions to the station. I assume that the German speaker is knowledgeable and willing to help me get to the station. The process succeeds if I *do* get to the station. I know whether the German question, "Wo ist der Bahnhof, bitte?" is the correct question to ask for my purposes, not by itself, but only by the success of the entire procedure: translation, answer in German, translation back to English, getting to the station.

But the writer of the translation manual has no idea whether I will want to get to the station or to the bathroom or will need a beer. Rather, the translation manual must anticipate all possibilities and convert an arbitrary English question into one in German which "asks the same thing." We know from Quine[12] that characterizing generally this notion of "asks the same thing" is problematic. But the compiler context allows us to be more specific about the difficulties. Suppose for simplicity that the program α implements some function f correctly, that is, given an input n, α produces an output $f(n)$. The compiled version of α is $C(\alpha)$. What does it mean to say that $C(\alpha)$ is correct, that is, that it also "computes" f?

What we need in order to use $C(\alpha)$ is a coding m of numbers into the kind of objects that machine language can manipulate. The number n will then be coded as the object $m(n)$. We ask that the compiled program work in such a way that, if its output on $m(n)$ is x, x should equal $m(f(n))$. That is, we should have

(1a) $m(f(n)) = [C(\alpha)(m(n))$

or equivalently,

(1b) $f(n) = m^{-1}[C(\alpha)(m(n))$

We can compute f by coding the input, compiling–which computes f–applying the compiled program $C(\alpha)$ to $m(n)$ and then applying m^{-1}, the inverse or the decoding operation of m, to the output of $C(\alpha)$.

A correctness proof for a compiler C consists of showing that for each program α, its compiled version $C(\alpha)$, *together with* the coding operation m, produces a mathematical object which is the denotational-semantic "meaning" of α. In other words, such a proof shows that conditions (1a) and (1b) hold.

Let us illustrate this apparently complex picture by means of an example. Consider the (simplified) Pascal program $\alpha = (x := 2;\ y := 3;\ z := x \times y)$, which means: make $x = 2$, make $y = 3$, and make $z = x \times y$. Suppose we represent the number n by the string a^n in the computer, where a^n stands for the symbol a repeated n times. Then $m(2) = aa$ and $m(3) = aaa$. Thus we will want the compiled version $\theta = C(\alpha)$ of the program α to store, first the string "aa" in the location

[12] W. V. Quine, "Meaning and Translation," in *The Structure of Language*, eds. J. Fodor and J. Katz (New York: Prentice Hall, 1964), pp. 460-478.

marked x, then the string "aaa" in the location marked y, and finally the string "$aaaaaa$" in the location marked z. Thus we can think of both α and $C(\alpha)$ as performing the multiplication of 2 by 3.

But note that the operational semantics of $C(\alpha)$ does not actually fix the denotational semantics of α. For example, under our code the number 0 is represented by the empty string. But we might have preferred that the nonempty string a represent the number 0, and that a^{n+1} represents the number n. In that case, we would have a coding m' with the property that $m'(1) = aa$ and $m'(2) = aaa$ So now aa represents the number 1, aaa represents 2, and $aaaaaa$ represents 5. The very same machine language program θ will now be interpreted as computing, not 2×3, but $g(1, 2)$ $(= 5)$ where g is the function $g(m, n) = (m + 1) \times (n + 1) - 1$. Also $g(2, 3)$ is not 6, but 11. Thus there is no unique answer to the question, "Which mathematical function is θ computing?" Rather it depends on the compiler C, the coding m, and the high-level program α, and cannot be recovered from θ alone.

Here we have a precise computer science analog to Quine's famed indeterminacy thesis. As Quine imagined it, an anthropologist landing on a remote island where a hitherto unknown language is spoken has a problem getting from the observed utterances of the natives to a translation manual for their language. This problem of "radical translation," of trying to learn the native language by observation alone, is exactly like the problem of determining the denotational semantics of a program α by watching the running of the compiled program $C(\alpha)$. We can figure out the denotational semantics of the original α only if α can be recovered, and this is possible only if the compiler C is known. But the anthropologist has no way of knowing the linguistic analog of the coding C. As Quine argues, when a native has responded to the appearance of a rabbit by saying "gavagai," we do not know whether he has referred to a rabbit, to a rabbit part, or perhaps to a rabbit seen while in the company of an anthropologist! Moreover, a child born into such a community is also in the position of the anthropologist. This child has no data except his or her observation of what people do. The child too, then, initially takes in language at the observational level. To "understand" what other people are saying requires obtaining the denotational semantics of various procedures and going on no more data than the anthropologist practicing radical translation. But what if we could actually look into the heads of people? Could we not then see what compiler is being used and recover both α and the intended meaning of whatever is being said? The answer to that is, No: such a procedure would still leave us at a syntactic level, albeit in a more detailed way. Even if we were given the formal expressions for the original α, the program text for the compiler C, and the formula for the coding m, in our role as radical translators we still would not know *what* mathematical objects the program referred to. For instance, there is nothing in the structure of the original α to indicate whether the function denoted by $\times$ is multiplication or some other function. In fact, if α_1 is the program $(x := 2); (y := x \times\ z)$, there is nothing in the notation even to indicate that the program is to read from the left to right rather than the other way round. It is our practice that we do read programs from left to right, but nothing in the notation itself forces us to do so, and nothing can.

This argument connects Quine's indeterminacy of translation thesis to a later argument offered by Kripke as an interpretation of Wittgenstein. In *Wittgenstein on Rules and Private Language*, Kripke presents arguments that seem to show that, even if there is a function *plus*, there is no way that we can unambiguously show that, when we use the symbol $+$, we are talking about the addition function and not some other function, for example *quus*, where $quus(x, y) = x + y$ if x and y are less than some very large number–for the sake of argument, 750–and $quus(x, y) = 5$ otherwise. No matter how much I say about my understanding of the word "plus," it is always possible to find some other function *quus* that is not addition and fits all that I or indeed anybody else has said about *plus*. Of course the function *quus* would need to be fitted to all that has been said about *plus* so far, so that the limit of 750 above might need to change, but there always *is* such a function. Parallel to this puzzle is this fact about denotational semantics: no matter how much we know about the operational semantics of a program, the ambiguity in the denotational semantics of the original high level program will remain. [13]

Should we allow ourselves to be moved to philosophical skepticism about meaning as a result of these sorts of arguments? Quine advocates only skepticism about observation-transcendent notions of meaning or propositions. In contrast, Kripke, on behalf of Wittgenstein, suggests that the plus-quus puzzle leaves us without any notion of a *fact* of meaning for an individual speaker. He claims that Wittgenstein's answer is to bring in the behavior of a wider linguistic community to settle questions of meaning. But matters will not stand on any more solid a basis by appealing to community usage. For that too is only "correct" relative to an antecedent purpose or specification of what correctness consists in.

Am I suggesting that human mental life in general and language in particular is nothing but computation? Not exactly; I am only drawing a parallel. The most influential recent argument that the human mind is not a computer has been advanced by Searle. Searle wishes to make an ambitious point: that a computer (and even a robot) cannot know, say, Chinese. I shall leave aside the merits of Searle's stronger claim to argue instead that we should grant him a weaker point. And that is that, *given certain restrictions*, neither Searle nor a computer in his specific position can be said to know Chinese. In his thought experiment, Searle imagines himself locked in a room having been given some instructions (in English) for correlating certain questions posed in Chinese with certain other expressions (stories) in Chinese so as to produce certain other expressions (the answers) in Chinese. If the rules are sufficiently good, then presumably Searle's written answers, conveyed out of the room, will be indistinguishable from those given by a Chinese speaker based on the latter's understanding of the stories. But Searle says that these rules do not suffice to allow him to say that he *knows* Chinese. And he argues that, since a computer that has been programmed to follow these rules knows no more than he does, then it too cannot be said to know Chinese.

What we have here is in fact an example of a compiler. A question asked in Chinese notation is compiled into a question asked in English, and the answer in English is decompiled into an answer in Chinese notation. English plays here the role of machine language and Chinese of the high-level language. And Searle is quite right that a person who can carry out this process need not know the "semantics" of Chinese.

As we have seen, however, there is a strong sense of the word "semantics" in which even a Chinese speaker cannot know the semantics of Chinese. She was born into a Chinese community and only has the same sort of data at her disposal that a radical translator would have. We might think, with Chomsky, that perhaps the semantics might be hardwired in her brain and need not be learned. But since humans are capable of learning very many languages with different semantics, the "hardwiring," if it exists, cannot settle all uncertainties about which semantics is intended by certain utterances and certain behavior.

Is there then a sense in which a Chinese person *does* know Chinese but in which Searle, in his example, does not? A crucial difference is that the Chinese speaking person can also correlate her behavior to the real world. If she wants five potatoes, she can go to a grocer, ask her question in Chinese, and see to it that what she gets is five potatoes and not five tomatoes. But note that, at

[13]Kripke himself suggests a connection with Quine's indeterminacy of translation thesis; see *Wittgenstein on Rules and Private Language*, pp. 14-15.

least in English, Searle's procedure cannot distinguish between potatoes and tomatoes.

To explain, suppose we have a procedure P for answering questions posed in English, correlated to situations described in English and yielding answers in English. And suppose that this procedure works properly. Now replace English by *Penglish*, in which the word "potato" means tomato and the word "tomato" means potato. The procedure P will also work perfectly for Penglish.

Consider the story, "Jane has no potatoes. John gives her five potatoes. Then he asks her to return three of the potatoes to him." And the question, "How many potatoes does Jane now have?" The answer is "two potatoes," which is clearly correct in English. But it is also correct in Penglish. In Penglish the story, the question, and the answer refer to tomatoes, using the Penglish word "potatoes," and the answer given by the procedure is still correct.

Knowing English implies knowing what potatoes are and that they are different from tomatoes. But someone who can answer questions like the above need not know the difference between English and Penglish and hence cannot necessarily be said to know English. Since a computer that has been programmed to follow the procedure also does not know the difference between a potato and a tomato, or between English and Penglish, we can say that it too does not know English. So Searle is quite right that merely knowing the rules for playing the question-answer game does not imply a knowledge of Chinese. If Searle is able to memorize his rules so that he can use them in real life with facility, however, and, moreover, if he is able to ask in Chinese for potatoes and then not accept tomatoes, it would be hard to say just what it is that the average Chinese person knows (in such contexts) that Searle does not. A robot that knew how to answer questions and was also able to operate with the language in the real world would be in the same position as this (more retentive) Searle. If we still wanted to say that the robot did not know Chinese, it would have to be on some other basis than Searle's Chinese room puzzle.

Returning to the program context, suppose we are given a compiler and a high-level language together with a denotational semantics for the high-level language. Suppose now that there is a property P that high-level language programs may have or fail to have. Then we could say–with some abuse of language–that $C(\alpha)$ has property P if and only if α has property P. But if some machine language program θ is *not* of the form $C(\alpha)$ for some α, then–under that particular compiler–it would not have a mathematical meaning, though it would still do *something*, that is, still have operational semantics.[14] But we could not say either that it had or that it lacked the property P. Returning to the linguistic context, I offer the following parallel: there need not be a determinate answer to the question whether the word "slab" in the language of Wittgenstein's builders in §2 of the *Philosophical Investigations* is a noun or a single word sentence. To answer such a question requires a great deal more information about the builders than a mere description of what they do with the word. In the linguistic context, what we are given is the linguistic behavior of individuals in a community, supplemented by non-linguistic behavior. As we noted, the latter is missing in Searle's Chinese room, which is significant. We want to develop a meaning theory, that is, a high-level language L with a denotational semantics that will "compile" into actual utterances. Communication between two members of a community, something that takes place at the lower level, could be thought of as conveying these high-level meanings. But developing such a high-level language and its semantics may not always be possible. In addition, it will almost certainly not be unique. Thus, assigning a meaning theory to the behavior of a linguistic community will be problematic, as will be the question (the radical translation question) of correlating the (supposed)

[14]This is perhaps the point of Piero Sraffa's famous gesture to Wittgenstein. The gesture that Sraffa made–brushing his chin with his fingertips–had a meaning, at least to Italians (or at least to Neapolitans), but could not fit into the semantics provided by Wittgenstein in the *Tractatus*. See Ray Monk's *Ludwig Wittgenstein: The Duty of Genius* (New York: Penguin, 1990), pp. 260-261.

meanings associated with the linguistic behavior of one community with the meanings associated with the behavior of another. Indeed–as Quine has argued–we even have to give up the picture of a conversation between two members of the same community as conveying meanings, for the meanings of the speaker cannot really be conveyed to the listener.[15]

Linguistic utterance is a social phenomenon that facilitates living in a community. It is undeniable that utterances *can* at times be interpreted in terms of a meaning theory–and such an interpretation, when it exists, yields significant benefits in terms of organizing our grasp of the workings of the language. (For example, first-order model theory, though not the only possible interpretation of first-order logic, makes it easier for us to understand the purely proof theoretic properties of first-order logic.) But such an interpretation may not always be available, or be unique, or be uniform over the different suburbs of the linguistic metropolis. The point of this observation is to suggest a shift in the way both computer scientists and philosophers should think about semantics.

[15]It is shown in my "Vagueness and Utility: The Semantics of Common Nouns," *Linguistics and Philosophy* 17 (1994): 521-35, and "Vague Predicates and Language Games," *Theoria* [Spain] XI/27 (1996): 97-107, though, that the listener will usually be *helped* by what he hears.